# Domain-Speculus Detailed specification

## 1. Purpose and Role

The purpose of the Domain-Speculus is to extract, from the user's free form text, the rules that the producer must obey. The Domain-Speculus is driven by the domain and interacts with the user in a round robin way to extract domain rules to use from the user's intent. To extract rules, it uses the Domain_Vocabulary.xml and the Predefined_Domain_Rules.xml copied inline into the ARCOS_Project.xml file sent with the request. The Domain Schema is also included into the ARCOS_Project.xml file.

Once the project has been defined, the Domain -Speculus is the first step of many. Through a free form text entry area, extract the possible business rules to add to the list of rules to obey. Each rule is also linked to the originating user intent as to see the original intent of each rule.

The Orchestrator sends NewDomainSpec (after ARCOS-Speculus completes) or EditDomainSpec; it may also forward ProducerClarification or ValidatorClarification.

When asked for clarification, the Domain-Speculus processes the request it got from the Orchestrator about a question from the producer or the validator. It tries to ask the user about the clarification to have an answer for the process and try to add rules appropriately.

The Domain-Speculus once all rules are accepted or rejected, and all user intents have been processed, it sends back the Domain_Rules.xml to the Orchestrator as to let the looping process to begin, sending the project file, the rule file to the producer and get a first trial of the output.

Cancelling at this point is equivalent to a <Cancelled> response.

If no rules were added. The user still has to accept the rules globally and send them to the Orchestrator. If he cancels, the process stops, sending a Project Cancelled to the user.

## 2. Process Flow

The first thing that the Domain-Speculus does is to read and add the Predefined Domain Rules to its rule roster.

Then displays on screen the current rules and their provenance (user intent, or predefined) and have a section to accept free-form text from the user and have a button to process it to extract possible rules and suggest them to the user to accept or decline the suggested rules it found in the user intent.

For each accepted rule, it links them both one to the other, the rule and the user intent as to keep track of the why of the rules.

On clarification requests from either the Producer or the Validator, the Domain-Speculus shows the request to the user and from his reply, try to add some new rules to help. If no rules are added the user still have to accept the current rules as to send them to the Orchestrator so the process can restart.

The user can then either cancel or save and to send the rules to the Orchestrator to start the loop process (Produce, Validate, Post-Process).

On Max Trials attained, the Orchestrator sends to the Domain-Speculus the last error report from either the Validator or the Post-Processor to try to extract from the reports some rules to suggest to add to the user. The user can reject or accept the proposed rules, and/or try to add new ones using free-form text from the user. Again. the user can then either cancel or save and send the rules to the Orchestrator to start the loop process (Produce, Validate, Post-Process).

All communications between the Orchestrator and the Domain-Speculus uses XML messages that need to respect the corresponding schema (XSD). The interaction with the user is left to the developer of the Domain-Speculus to define.

The Domain-Speculus serves to define the project's rules to use in the orchestration loop.

## 3. Communication between Domain-Speculus and the Orchestrator (Maestro) using schema-based XML.

The Maestro_Domain_Speculus_Messaging.xsd has the following elements.

| | | |
|---|---|---|
| <DomainSpeculusMessage> | | |
| <NewDomainSpec> | Choice | |
| <ProjectID> | | Required |
| <EditDomainSpec> | Choice | |
| <ProjectID> | | Required |
| <ProducerClarification> | Choice | |
| <ProjectID> | | Required |
| <Question> | | Required |
| <ValidatorClarification> | Choice | |
| <ProjectID> | | Required |
| <Question> | | Required |
| <GotToMaxTrials> | Choice | |
| <ProjectID> | | Required |
| <DomainSchema> | | Required |
| <RulesXML> | | Required |
| <ProducerOutputZip> | | Required |
| <LastReportXML> | | Required |
| <LastReportKind> | | Required |
| <Checksum> | common:SHA256Type | Required |

The Domain_Speculus_response.xsd has the following elements.

| | | |
|---|---|---|
| <DomainSpeculusResponse> | | |
| <Success> | Choice | |
| <ProjectID> | | |
| <SavedAt> | Location | |
| <RulesXML> | | |
| <RuleCount> | | |
| <IntentCount> | | |
| <Timestamp> | | |
| <Cancelled> | Choice | |
| <ProjectID> | | |
| <Reason> | | optional |

| | | |
|---|---|---|
| <Timestamp> | | |
| <Checksum> | common:SHA256Type | Required |

# 4 Elements and Attributes of the Domain_Vocabulary.xsd

| | | |
|---|---|---|
| <Vocabulary> | | |
| @name | | required |
| @version | | optional |
| <Entities> | | |
| <Entity> | | |
| @name | | |
| <Synonym> | | |
| <Attributes> | | |
| <Attribute> | | |
| @name | | |
| <Target> | | |
| <Synonym> | | |
| <Enums> | | |
| <Enum> | | |
| @name | | |
| <Value> | | |
| <Units> | | |
| <UnitFamily> | | |
| @name | | |
| <Unit> | | |
| @name | | |
| @abbr | | |
| @toBase | | |
| @isBase | | |
| <Phrases> | | |
| <Phrase> | | |
| @ruleType | | |
| @appliesToTarget | | |
| @CrudOp | | |
| @Scope | | |
| <Text> | | |

## 5. Elements and Attributes of the Predefined_Domain_Rules.xsd

| | | |
|---|---|---|
| <PredefinedDomainRules> | | |
| @name | | |
| @version | | Optional |
| <Rules> | | |
| <Rule> | | |
| @id | | |
| @target | | |
| @type | RuleTypeEnum (see below) | |
| @predefined | true | |

The RuleTypeEnum has the following values in it.

| |
|---|
| required |
| enum |
| reference_exists |
| unique_global |
| has_components |
| match_pattern |
| range |
| equal_to |
| Custom |

## 6. Elements and Attributes of the Domain_Rules.xsd

| | | |
|---|---|---|
| <Specification> | | |
| @name | | |
| @version | | Optional |
| <ProjectID> | | |
| <Rules> | | |
| <Rule> | | |
| @id | xs:ID | |
| @target | | |
| @type | RuleTypeEnum (see below) | |
| @predefined | True | |
| <Description> | | |
| <Origin> | | |
| @ref | xs:IDREF | |
| <Intents> | | |
| <UserIntent> | | |
| @id | xs:ID | |
| <RawTect> | | |
| <RulesExtracted> | xs:IDREF | |

# 7. How to use the Domain_Vocabulary.xml, Predefined_Domain_Rules.xml to extract Domain_Rules.xml

**How the developer uses the two schemas to extract rules**

**0) Load the domain knowledge**

1. Load **Predefined_Domain_Rules.xml** → a set of baseline rules (e.g., required, unique_global, reference_exists, etc.). Treat these as already accepted, unless the user explicitly overrides them.
2. Load **Domain_Vocabulary.xml** →
    - **Entities** + **Attributes** (+ **Synonym**) let you do robust NER/slot-filling: "part", "item", "SKU", "stock keeping unit" all resolve to the same target.
    - **Enums** (valid values) support value normalization and validation.
    - **Units** (families, base unit, toBase) allow numeric normalization ("50 kg" = base-unit value).
    - **Phrases**: reusable patterns with metadata like @ruleType, @appliesToTarget, @CrudOp, @Scope. These are the shortcuts that map user text to rule classes without re-inventing regex/LLM prompts each time.

Why this matters: ARCOS expects the **Domain-Speculus** to produce Domain_Rules.xml and link each rule back to the source **UserIntent** (traceability). Vocabulary + predefined rules are the scaffolding to do that reliably.

**1) Parse user free text into intents**

- Split the conversation into **UserIntent** chunks (per paragraph/sentence or per "Submit" click).
- For each intent, keep the raw text and an ID for traceability (you'll refer to it from each proposed rule). This is the linkage your docs call out explicitly.

**2) Resolve entities/attributes/synonyms**

- Use Vocabulary/Entities/Entity[@name]/Synonym and Vocabulary/Attributes/Attribute[@name]/Synonym to canonicalize mentions ("SKU", "stock code" → Part.sku).
- Use Attributes/Target to attach the attribute to its owning entity (e.g., sku belongs to Part).

**3) Detect rule intent via Phrases**

- Match text against Vocabulary/Phrases/Phrase entries. Each phrase advertises:
    - @ruleType → maps directly to your **RuleTypeEnum** (e.g., required, unique_global, range, match_pattern, enum, equal_to, reference_exists, has_components, Custom).
    - @appliesToTarget → which entity/attribute this phrase governs.
    - @CrudOp → if relevant (Create/Read/Update/Delete).
    - @Scope → record/collection/global scope (e.g., "globally unique across all Parts").

This lets you convert "Every part must have a unique SKU" into a **unique_global** rule targeting Part.sku with scope global, without inventing ad-hoc parsing every time.

**4) Normalize values with Enums and Units**

- If the phrase implies ranges or enumerations:
    - Use Enums/Enum[@name]/Value to see if the mentioned value is valid.
    - Use Units/UnitFamily[@name]/Unit[@name/@abbr/@toBase/@isBase] to normalize numbers to base units ("between 0.1 kg and 50 kg" → store base-unit min/max).
- If a value doesn't exist in the vocab, propose a **Custom** rule and ask for confirmation (or add the value to the vocab later).

**5) Build proposed rules (don't commit yet)**

For each detected constraint, build a candidate **Rule** object:

- id: temporary UUID (finalized when accepted)
- type: from @ruleType (RuleTypeEnum)
- target: from @appliesToTarget (e.g., Part.sku, Part.weight)
- Any parameters the type needs (e.g., min/max for range, pattern for match_pattern, enumRef for enum)
- predefined: false (since it's a new suggestion)
- userIntentRef: link to the intent that caused it (traceability)

Then **diff** each proposal against **Predefined_Domain_Rules** so you don't duplicate or contradict existing constraints. If you detect conflicts (e.g., predefined says weight is required but user says "weight optional"), surface as **Conflict** for human decision.

**6) Present to the user for Accept / Edit / Reject**

- Show grouped by **Entity → Attribute → Proposed Rules**.
- Highlight **New / Update / Conflict** status (your docs call out this review loop).
- On acceptance, assign stable IDs and mark predefined=false. On edits, update parameters (e.g., change range max).

**7) Emit Domain_Rules.xml (plus UserIntents)**

- Produce **Domain_Speculus_Response (Success)** carrying Domain_Rules.xml (and keep the mapping to **UserIntents**). The orchestrator now proceeds to Producer/Validator, exactly as in your lifecycle docs.

## 8-Concrete mini-example (BLEU parts)

**User says:**
"Every Part must have a unique global SKU and a weight between 0.1 kg and 50 kg."

**Vocabulary hits:**

- Entities/Entity name="Part" (+ synonyms "item", "component")
- Attributes/Attribute name="sku" (synonyms "stock code", "stock keeping unit"), Target=Part
- Attributes/Attribute name="weight" (Target=Part), Units family "mass" with kg toBase=1000 (say base=grams)

**Phrase matches:**

- Phrase A: @ruleType="unique_global" @appliesToTarget="Part.sku" @Scope="global"
- Phrase B: @ruleType="range" @appliesToTarget="Part.weight"

**Proposed rules:**

1. unique_global on Part.sku (scope=global)
2. range on Part.weight with min=100 and max=50000 in base grams

Diff against **Predefined_Domain_Rules** (if you already have unique_global for Part.sku, mark as "Already covered"; otherwise propose as New). User accepts → they're written to **Domain_Rules.xml** and linked back to the intent text. This is precisely the "rules linked to original user intent" story in your docs.

---

**Where this fits in ARCOS**

- Maestro ↔ Domain-Speculus messaging is **schema-defined** (New/Edit, Clarifications, Checksum), and Domain-Speculus returns Success/Cancelled. Your Domain layer is the **only** place where clarifications about rules happen; ARCOS-Speculus never gets those.
- After acceptance, the Orchestrator continues the loop with Producer → Validator → Post-Processor, using your rules and schema.

---

**TL;DR for the developer**

- **Vocabulary** = your *recognizer & normalizer* (entities, attributes, synonyms, enums, units, canned phrase→rule mappings).
- **Predefined rules** = your *baseline constraints* (seeded once, never spam the user about them).
- **Algorithm** = intent chunk → vocab resolve → phrase→rule mapping → normalize values → diff against predefined → propose → accept/edit/reject → emit Domain_Rules.xml with links back to intents.

The Domain_Vocabulary.xml file contains the following elements and structure:

Entity, with synonyms

Attributes,
target = one Entity

Enums

Units, with base and scaling factors

Phrases,
ruleType = RuleTypeEnum,
Applies to target = Entity.Attribute,
CrudOp = Create,
Scope = record or global

The Predefined_Domain_Rules.xml file contains the following elements and structure

Rule,
target = Entity.Attribute
type = RuleTypeEnum
predefined = true

The Domain_Rules.xml obtained from the Domain-Speculus has the following elements and structure

ProjectID

Rules
Rule
id
target = Entity.Attribute
type = RuleTypeEnum
predefined = false
Intents
UserIntent
RawText
RuleExtracted